Preprint
UCRL-JC-134242

Qualification of Safety-Related Software in Nuclear Power Plants

J. D. Lawrence and G. L. Johnson

*This article was submitted to
International Topical Meeting on Nuclear Plant Instrumentation,
Controls, and Human-Machine Interface Technologies
Washington, DC
November 16-18, 2000*

June 13, 2000

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

International Topical Meeting on Nuclear Plant Instrumentation, Controls, and Human-Machine Interface Technologies (NPIC&HMIT 2000), Washington, DC, November, 2000.

Qualification of Safety-Related Software in Nuclear Power Plants

J. Dennis Lawrence and Gary L. Johnson
Lawrence Livermore National Laboratory
Livermore, California USA 94550
lawrence2@llnl.gov johnson27@llnl.gov

Keywords: Safety system software, flowgraphs, safety qualification

ABSTRACT

Digital instrumentation and control systems have the potential of offering significant benefits over traditional analog systems in Nuclear Power Plant safety systems, but there are also significant difficulties in qualifying digital systems to the satisfaction of regulators. Digital systems differ in fundamental ways from analog systems. New methods for safety qualification, which take these differences into account, would ease the regulatory cost and promote use of digital systems. This paper offers a possible method for assisting in the analysis of digital system software, as one step in an improved qualification process.

1. Motivation

Digital instrumentation and control (I&C) systems have the potential of offering significant cost, performance and reliability benefits over traditional analog systems. When used in safety-critical applications, such as nuclear power plant (NPP) safety systems, such I&C systems must be qualified to meet concerns of regulators, users and the public. For computer-based safety-system components, this qualification includes the design qualification of the embedded software components to confirm that the component can be trusted to perform its safety functions under the full range of operating conditions.

Current state-of-the-practice in software engineering includes no single method that can be used to qualify NPP safety software. The United States Nuclear Regulatory Commission (NRC) practice, as documented in the Standard Review Plan (NRC, 1997), focuses attention on the process used to develop the software, in the belief that a well-qualified development process is more likely to create satisfactory software than a poorly-qualified process. This attention is supplemented by thorough verification and validation of the design outputs produced by the development process. This is an expensive process. Industry estimates show that verification and validation accounts for 20 to 40% of the development cost in a well designed development program. In poor development programs the cost can be much higher. This project was triggered by the desire to find less expensive ways for software qualification.

The idea of making a radical improvement in qualification methods for the general case of safety software is daunting. For this work we chose to focus on simple digital components of safety systems. Many NPP safety systems components have characteristics similar to those of many other process control safety applications – personnel protection, effluent waste monitoring, building ventilation, consumer products, and the like. These systems (termed here *small safety systems*) tend to be small (thousands of lines of code, not millions), have limited inputs from sensors, have limited outputs to actuators, and have restricted functionality focussed on detection of unsafe process conditions and safe shut down. There are a great number of such systems – indeed, it is likely that many people in the United States is “touched” by several small safety systems everyday, given their

This work was performed under the auspices of the U.S. Department of Energy by the University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

prevalence in microwave ovens and other consumer devices. Nuclear power plant examples of such components might include panel meters, protective relays, time-delay relays, diesel speed controls, and diesel load sequencers.

Qualification of analog components has a long history, and is reasonably well understood. The necessary confidence is often developed through a combination of design evaluation, testing and inspecting the equipment, and operating experience review (EPRI, 1988). This way of developing confidence is possible because (1) analog components are built from standard modules with known properties, (2) design documents are available and described in a well understood language (such as schematic diagrams), (3) the performance is constrained by physics, and (4) physics models exist to predict the performance. These attributes permit the designer to construct a model based upon physics principles that will predict discontinuities in the system transfer function. One can then conclude that the instrument will react properly throughout the range of input values based on limited testing.

Unfortunately, this approach will not work for digital components because in most cases (1) the software in these systems is not constructed using standard modules, (2) software design information is frequently unavailable, and (3) software functions are not constrained by physical laws. Therefore, it is difficult to construct practical design models to support the use of testing and operating experience for design qualification.

At the present time, the developer must attempt to convince the licensee or the regulator that the software has no safety-significant failure modes by means of an argument which is intended to develop confidence in equipment by gaining confidence in the developer's design, verification and validation activities. This argument typically includes extensive testing and the use of a high-quality development process. We would like to have a viable alternative to this form of argument for a number of reasons. First, safety-significant failures are frequently the result of unexpected combinations of rare events which may well escape notice in testing. Second, process assessment does not directly measure the safety of the product of interest, but instead relies on an inference about safety from observations of the development process. Third, process assessment requires labor-intensive inspections of development processes and records, and often cannot be carried out on existing systems when the development records are not available. A more appealing argument in support of a safety conclusion would depend on inferences that are easier to support .

The Lawrence Livermore National Laboratory (LLNL) has been engaged in a small research effort during 1998 and 1999 attempting to develop an alternative approach to qualification of simple safety components by combining knowledge of the properties of such components with limited testing, and then combining the results with knowledge of other properties of the component to permit a reasoned argument in support of qualification. This paper is concerned with a portion of the general problem – analysis of the software itself that can be used to draw conclusions about its properties. The result is to enable construction of an argument that is <u>analogous</u> to the continuity argument used for analog systems.

2. Simple Systems

The initial motivation for our research was consideration of the implications of a number of articles on software testing. Theoretical arguments by Littlewood (1993) Bulter (1993) and Poore (1993) imply that enormous amounts of testing are required for modest assurance of software reliability. While attempting to understand the implication of these arguments, we concluded that they are based on assumptions of ignorance. That is, the arguments assume no knowledge of the structure of the software or its input space. We concluded that if knowledge of the software and its input space could be made available, then one ought to be able to construct a more optimistic argument. For simple components, the input space is generally very simple and is nearly always well understood. Therefore, the interesting question is obtaining knowledge of the software itself.

It is generally agreed that safety systems and components should be kept simple, but there is no widespread agreement on exactly what this means when applied to software. We believed that "simple" should be defined in such a way as to permit careful analysis of the software. We therefore began by deciding how we wished to analyze the software, and then determined what types of simplification rules needed to be applied to the software in order to make the analysis feasible. That is, we adopted a pragmatic approach of imposing design rules on the software that makes possible the analysis we wanted to perform. Somewhat to our surprise, these rules to not appear to be especially burdensome on the programmers.

The software analysis we chose is based on the theory of flowgraphs and the theory of software testing.

3. Theory

A *flowgraph* is a directed graph in which the nodes represent instructions (or small groups of instructions) and the edges represent potential transfers of control. In keeping with the usual theory of flowgraphs (Hecht, 1977; Muchnick, 1981; and Fenton, 1985) we assume the existence of a unique start node from which every other node in the flowgraph can be reached by some path, and a unique stop node which can be reached from every other node in the graph. A path in this graph from the start node to the stop node is an abstraction of a possible execution path in the software itself. Thus, analysis of the software execution can be carried out by analyzing all the execution paths in the flowgraph.

Fig. 1 shows a portion of a program taken from (White, 1980), and the resulting flowgraph is shown in Fig. 2. In this diagram, nodes are shown as circles and directed edges by arrows. Labels in the nodes represent the statements and conditions in the program.

The information in the flowgraph is sufficient to construct a tree, termed the *flowgraph tree*, whose branches are the potential execution paths of the program. In general, this tree will be infinite. The tree form can theoretically be used to determine the exact conditions that will cause each branch to be executed, and to determine the functions that will be calculated on each branch. Fig. 3 shows the flowgraph tree for the example.

```
read I, J ;
if I ≤ J + 1
        Then K = I + J – 1 ;
        Else K = 2 * I + 1 ;
if K ≥ I + 1
        then L = I + 1
        else L = J – 1 ;
if I = 5
        then M = 2 * L + K ;
        else M = L + 2 * K – 1 ;
write M ;
```

Fig. 1 Example Portion of a Program

The computer science literature has shown that there are four classes of potential faults in program, termed domain faults, missing predicate faults, computational faults and timing faults. The flowgraph tree can be analyzed in a reasonably straightforward manner to determine if any potential faults in these classes exist for the branch. In potential faults do exist for a branch such that unsafe failure is possible, then qualification of that branch can be rejected. If such do not exist, then the digital system may be qualified for that branch. Extending this to all branches can be used to infer the potential for, or lack of, any unsafe failure modes in the program.

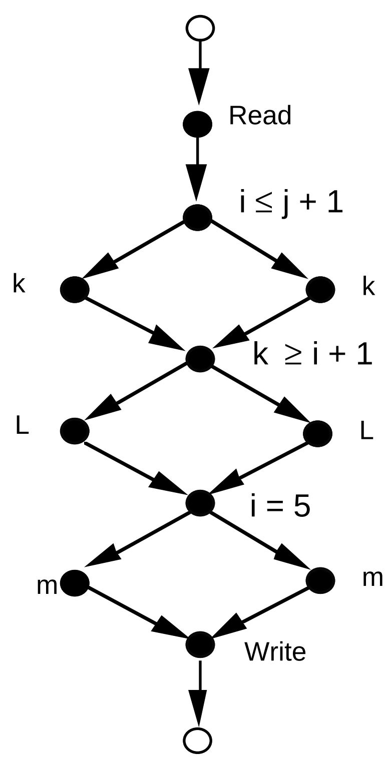

Fig. 2. Flowgraph for Fig. 1

4. Design Rules

It now became necessary to impose a set of rules, termed *design rules*, on the software is a small safety system so that the analysis outlined in the last section is possible. This takes several forms: fundamental restrictions, general rules and specific rules.

4.1. Fundamental Rules

There are three fundamental restrictions that are assumed for a small safety system. First, it will be assumed that all input to the program comes from sensors and all output goes to actuators. In particular, the safety action is carried out independent of human operators.

Second, it is assumed that the safety assertion is known precisely as a mathematical relation on the sensor input values and actuator output values. For example, in a microwave oven, the safety rule is that either the power is off or the door is closed.

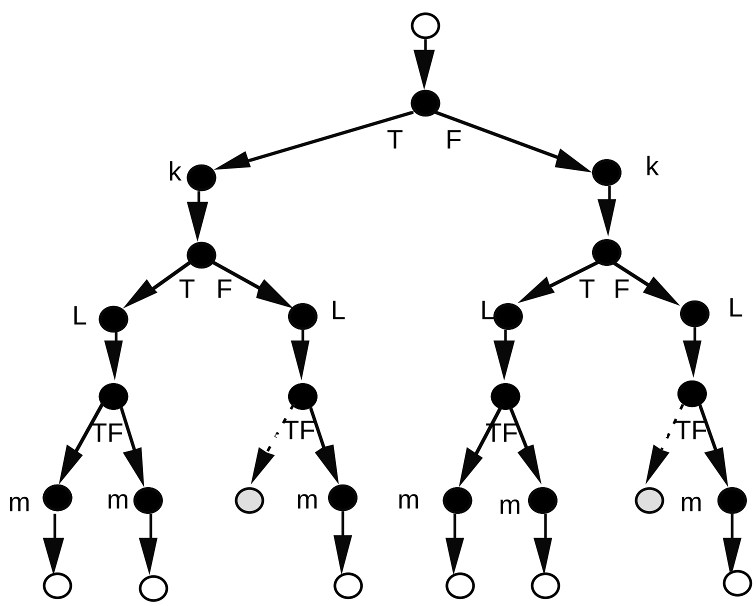

Fig. 3. Flowgraph Tree for Example in Fig. 2

Third, it is assumed that the program operates under a cyclic executive system. This causes the program to operate as a single repetitively executed program; the executive starts the program periodically, the program carries out its calculations and sets any necessary actuator results, and this process repeats indefinitely.

4.2. General Rules

There are three general rules that must be satisfied to carry out our analysis method. Each of these is phrased as a necessary condition on the software. They are:

- Flowgraph creation. It must be possible to create a flowgraph that completely represents the execution of the program. That is, the flow of control given in the flowgraph is presumed not subject to operating system interruptions and can be statically determined.

- Flowgraph tree creation. It must be possible to transform the flowgraph into a finite flowgraph tree.
- Fault detection. It must be possible to determine if any of the different classes of faults exist.

4.3. Specific Rules

The general rules are somewhat abstract, so more specific rules are required such that if the specific rules are obeyed, the general rules are satisfied. We have created a list of specific rules, which we believe are sufficient to satisfy the general rule; proof of this is left for future research. There are seventeen of these rules. It is worth noting that none of these rules make it impossible to write the software for a small safety system. The combination of these rules is our definition of a simple software system. The rules are listed briefly below; see (Lawrence, 2000) for justification of the rules and a more complete discussion.

- There shall be no interrupts, except for timer interrupts that occur at fixed intervals and have a known maximum duration.
- The operating system shall not perform time slicing.
- There shall be no multi-processing and no multi-tasking.
- There shall be no dynamically variable program structures, such as a go-to statement where the address is a variable.
- Each loop within the program must have an upper bound on the number of repetitions, where this bound is independent of any program input value.;
- There shall be no unbounded recursion.
- The input domain must be finite.
- The program transfer function must be a total function.
- The program transfer function must be known and computable.
- The maximum size of every program data structure must be fixed and finite.
- There shall be no dynamic memory allocation.
- Variant structures (termed “unions” in C) without tags shall be forbidden.
- All procedure and function calls must be protected by a prototype declaration supported by strict use of data typing.
- All variables shall be declared and typed.
- Unspecified, undefined and implementation-defined programming language feature must be avoided.
- Minimum and maximum timing requirements for each node in the flowgraph must be computable.
- Pointer arithmetic is forbidden and only one level of indirection is allowed.

5. Program Analysis

Even with the design rules, a small safety system can generate a very large number of branches in the flowgraph tree. Analyzing all of the branches with a reasonable cost requires that most of the branches be handled in an automated way. A prototype tool was constructed for this purpose, to explore the feasibility of analyzing each branch for possible safety violations. It is not to be expected that all branches can be completely analyzed automatically, but if the number of branches left for human examination can be kept limited, the expense of the process can be kept bounded. The success of the initial prototype has encouraged us to expand it to cover more of the analysis process.

The tool includes five main steps.

1. Create the flowgraph from the small safety system program
2. Verify that the design rules have been met
3. Transform the flowgraph into a flowgraph tree
4. Analyze each branch of the tree
5. Construct a safety argument from the analysis

The first three steps present no conceptual difficulties. There are, however, a number of issues that have to be resolved in order to carry out the analysis. This is sketched next; see (Lawrence, 2000) for more details.

- Each branch of the flowgraph tree can be distinguished from all other branches by means of the predicates that cause the branch to be executed. Thus, the branch consists of a mixture of predicates, each evaluated to TRUE or FALSE, and a sequence of other statement. All of these statement can be rewritten in terms of input and state variables using well-known methods (White 1980).
- In general, some of these predicates will be inconsistent: a branch, for example, may contain two predicates that imply $x > 0$ and $x < 0$. Such impossible branches can be eliminated from further analysis; the program will never execute them.
- Each assignment statement on a branch may be rewritten in terms of input and state variables. The result is a set of functions computed by the branch. These are termed the *branch functions*.
- It is now easy to determine that each variable on the branch is defined prior to use.
- Since it is assumed that the time requirements for each node of the branch can be computed, timing analysis for the branch is straight-forward.
- Bounds can be placed on the size of all calculated variables on the branch, using range arithmetic (Alefeld, 1983). This is sufficient to check for divide-by-zero and overflow possibilities.
- Array manipulations, subroutines, floating point arithmetic, and other matters require special attention. Research continues on these items
- Errors in compilers, operating systems, software libraries and the like must be covered by other means

This analysis, applied to the example flowgraph tree in Fig. 3, is shown in Table 1. Here, the branches of the tree are indicated by truth values for the three conditions in the three branch statements. Each condition can be written in terms of the input variables *I* and *J*, as shown. Likewise, each of the program variables *K*, *L* and *M* can be written in terms of the input variables. Two of the potential branches in the tree are not possible since the conditions for those branches are inconsistent.

Table 1. Predicates and Path Functions for Example Program in Fig. 3

	$i \le j + 1$	$k \ge i + 1$	$i = 5$	k	L	m
TTT	$i \le j + 1$	$j \ge 2$	$i = 5$	$i + j - 1$	$i + 1$	$3i + j + 1$
TTF	$i \le j + 1$	$j \ge 2$	$i \ne 5$	$i + j - 1$	$i + 1$	$3i + 2j - 2$
TFT	$i \le j + 1$	$j < 2$	$i = 5$	This path domain is nonexistent since it implies i = 5 and i ≤ 3.		
TFF	$i \le j + 1$	$j < 2$	$i \ne 5$	$i + j - 1$	$j - 1$	$2i + 3j$
FTT	$i > j + 1$	$i \ge 0$	$i = 5$	$2i + 1$	$i + 1$	$4i + 3$
FTF	$i > j + 1$	$i \ge 0$	$i \ne 5$	$2i + 1$	$i + 1$	$5i + 2$
FFT	$i > j + 1$	$i < 0$	$i = 5$	This path domain is nonexistent since it implies i < 0 and i = 5		
FFF	$i > j + 1$	$i < 0$	$i \ne 5$	$2i + 1$	$j - 1$	$4i + j$

6. Future Work

The next step in this research is to create a new prototype that uses the knowledge gained from the first one to automate more of the analysis, and handle some of the cases that the initial prototype cannot. The ultimate goal of the research is to be able to take an execution package (for example, a COTS software system), re-engineer this into a form that can be analyzed, and carry out the analysis. If this could be done, the dependence on compilers and operating systems could also be eliminated. There is, however, a long way to go before this possibility is more than a hope.

ACKNOWLEDGMENTS

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

REFERENCES

U.S. Nuclear Regulatory Commission, 1997. *Standard Review Plan for the Review of Safety Analysis Reports for Nuclear Power Plants*, Chapter 7, Rev. 4, NUREG-0800.

EPRI, 1988. *Guideline for the Utilization of Commercial Grade Items in Nuclear Safety Related Applications*. Palo Alto CA: Electric Power Research Institute, EPRI NP-5652.

Hatton L., 1995. *Safer C*, New York: McGraw Hill.

Littlewood B., Strigini L., 1993. Validation of ultrahigh dependability for software-based systems technique. *Comm. ACM* **36**, 69-80.

Butler R.W., Finelli G.B., 1993. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Trans. Soft. Eng.* **19**, 3-12.

Poore J.H., Mills H.D., Mutchler D., 1993. Planning and certifying software system reliability. *IEEE Software* **10**, 88-99.

Hecht M.S., 1977. *Flow Analysis of Computer Programs*, Amsterdam: North-Holland.

Muchnick S.S., Jones N.D. (Ed), 1981. *Program Flow Analysis*, Englewood Cliffs NJ: Prentice Hall.

Fenton N.E., Whitty R.W., Kaposi A.A., 1985. A generalised mathematical theory of structured programming. *Theoretical Computer Science* **36**, 145-171.

Lawrence, J.D., 2000. Software Qualification in Safety Applications. *Rel. Eng. and System Safety*, to appear.

White L.J., Cohen E.I., 1980. A domain strategy for computer program testing. *IEEE Trans. Soft. Eng.* **6**, 247-257.

Alefeld G., 1983. *Introduction to Interval Computations*, Academic Press.

Littlewood B., Fenton N., 1996. Applying Bayesian belief networks to system dependability assessment. In: *Proc. Safety Critical Sys. Club Symp*.